Latency-aware Leader Election *

Nuno Santos†
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
nuno.santos@epfl.ch

Martin Hutle
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
martin.hutle@epfl.ch

André Schiper
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
andré.schiper@epfl.ch

ABSTRACT

Experimental studies have shown that electing a leader based on measurements of the underlying communication network can be beneficial. We use this approach to study the problem of electing a leader that is eventually not only correct (as captured by the Ω failure detector abstraction), but also optimal with respect to the transmission delays to its peers. We give the definitions of this problem and a suitable model, thus allowing us to make an analytical analysis of the problem, which is in contrast to previous work on that topic.

Categories and Subject Descriptors

C.2.4D [**Computer-Communication Networks**]: Distributed Systems

Keywords

Analytical Analysis, Leader Election, Distributed Algorithms

1. INTRODUCTION

Leader election is an important service for fault tolerant systems. Such a service is typically used in several consensus algorithms, the so-called "leader-based" consensus algorithms. In such algorithms the leader can be determined either by a static rule (e.g., rotating coordinator), or can be computed dynamically. In the latter case the leader is selected by a distributed leader election algorithm. Leader election is also sometimes called "implementation of the Ω failure detector". A lot of work on the topic has been published in the recent years, e.g., [1, 2, 3, 6, 4]. One of the goals addressed was to weaken as much as possible the synchrony assumptions required for leader election. Another goal was to have an algorithm with two important properties: (i) stability and (ii) low message complexity. Stability means that the current leader is never demoted in favor of a new leader without reason. Low message complexity refers to the messages needed to monitor the leader. If the system consists of n processes, we would like ideally the monitoring to require only $O(n)$ messages. All these approaches ignore message latencies.

If the latencies of the different links in the system differ significantly, it might make sense to elect a leader that has links with low latency to the other processes in the system. This is typically the case for leader-based consensus algorithms in which the communication pattern is 1 to n, *i.e.*, communication is only between the leader and the other processes. Indeed in this case, choosing a leader with low latencies to other processes increases the efficiency of the leader-based consensus algorithm.

The idea of taking link latencies into account for leader election has been considered in [9, 7, 8]. In [9], an adaptive consensus algorithm is given. The initial coordinator is chosen to be a "fast" process based on measurements of the time each process takes to answer when it is a coordinator. In [7], the notion of "process order" is used to convey information about the link latencies. For a consensus algorithm based on $\diamond S$ and the rotating coordinator paradigm, the idea is to use the process order (instead of the process id) for selecting the successive coordinators (which requires global agreement on the matrix of latencies). For a consensus algorithm based on Ω, the process order is used within the implementation of Ω. However, the paper does not describe the implementation of Ω. In [8], a predictor is used for estimating the best process, but the stochastic model in which this predictor is supposed to work is not specified.

The contribution of the paper is to extend the above ideas by considering a latency-based system model, which is parameterized with a constant ε. We assume that during a stable period the latency of every link deviates only by ε from some (unknown) fixed value (where the value can be different for every link). The model allows an analytical analysis of the leader election algorithm (in contrast to the above best-effort approaches), and allows us to show that the leader election algorithm guarantees that during a stable period the leader latency to a majority of processes is optimal within 12ε. The optimal latency "to a majority" is related to the fact that, with benign faults, leader based consensus algorithms progress whenever the leader receives replies from a majority.

*Research funded by the Hasler Foundation under grant number 2070.
†This author was partially funded by the Portuguese Foundation for Science and Technology (FCT) (SFRH/BD/17276/2004).

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

2. SYSTEM MODEL AND PROBLEM DEFINITION

2.1 Model

We consider a model with n processes Π, which communicate over a fully connected network via message passing. Up to $f < n/2$ processes may fail by crashing. We assume each process has a local clock with no skew[1] and that computational steps take no time. The timing behavior of the link from process i to process j is described by a function $\delta_{ij}(t)$, that is, if a message is sent from i to j at time t, it is received at time $t+\delta_{ij}(t)$. If at time t process i has crashed, we define $\delta_{ij}(t) := \infty$. Since a process sends at most one message to another process at a certain time, this function is well-defined. If no message is sent at a time t, $\delta_{ij}(t)$ has no meaning, but to simplify presentation we assume that it has the same value as the last time i sent a message to j.

We define further the function $rtt_{ij}(t) := \delta_{ij}(t) + \delta_{ji}(t + \delta_{ij}(t))$, which intuitively models the round trip delay of a message if the peer process algorithm responds immediately to this message. Additionally, $RTT_i(t)$ denotes the vector of $rtt_{ij}(t)$ for a process i, sorted by ascending order of values. Finally, we define $majrtt_i(t) := RTT_i(t)[\lfloor n/2 \rfloor + 1]$, which models the time required for a two-way message exchange between i and a majority of processes.

We assume that the transmission delays in our system eventually stabilize. In more detail, we assume that there is a time GST, so that for all times $t \geq GST$ the following holds:

- All faulty processes have crashed.
- There is an a priori known value Δ, s.t. for each pair of correct processes i and j, $\delta_{ij}(t) \leq \Delta$.
- All message transmission delays remain within a window of 2ε, *i.e.*, $\forall i, j\ \exists \overline{\delta}_{ij} :\ \left|\delta_{ij}(t) - \overline{\delta}_{ij}\right| \leq \varepsilon$.

For a given run, for each link, fix one $\overline{\delta}_{ij}$ that satisfies the third condition. Then, in compliance with the definitions above, we can define $\overline{rtt}_{ij} := \overline{\delta}_{ij} + \overline{\delta}_{ji}$; $\overline{RTT}_i$ as the vector of $\overline{rtt}_{ij}$ for process i, sorted by ascending order of values; and finally $\overline{majrtt}_i := \overline{RTT}_i[\lfloor n/2 \rfloor + 1]$.

We now show that after GST, the round-trip time between two processes i and j remains within a window of 4ε centered around $\overline{rtt}_{ij}$:

LEMMA 1. *For $t > GST$, we have $\left|rtt_{ij}(t) - \overline{rtt}_{ij}\right| \leq 2\varepsilon$ and $\left|majrtt_i(t) - \overline{majrtt}_i\right| \leq 2\varepsilon$.*

PROOF. From the definitions we get $\left|rtt_{ij}(t) - \overline{rtt}_{ij}\right| = \left|\delta_{ij}(t) + \delta_{ji}(t + \delta_{ij}(t)) - \overline{\delta}_{ij} - \overline{\delta}_{ji}\right| \leq \left|\delta_{ij}(t) - \overline{\delta}_{ij}\right| + \left|\delta_{ji}(t + \delta_{ij}(t)) - \overline{\delta}_{ji}\right| \leq 2\varepsilon$.

For the second result, let $p_1, p_2, \ldots, p_n$ be the ordered list of processes corresponding to $\overline{RTT}_i$, that is, the processes ordered by increasing $\overline{rtt}_{ij}$ from i; let $q_1, q_2, \ldots, q_n$ be the similar list for $RTT_i(t)$, and let $m = \lfloor n/2 \rfloor + 1$. By definition, $\overline{rtt}_{ip_1} + 2\varepsilon \leq \overline{rtt}_{ip_2} + 2\varepsilon \leq ... \leq \overline{rtt}_{ip_m} + 2\varepsilon$. Since after GST, $rtt_{ij}(t) \leq \overline{rtt}_{ij} + 2\varepsilon$, we have $rtt_{ip_k}(t) \leq \overline{rtt}_{ip_m} + 2\varepsilon$, for all k, $1 \leq k \leq m$, and thus $majrtt_i(t) \leq \overline{majrtt}_i + 2\varepsilon$. A similar argument can be used to show that $majrtt_i(t) \geq \overline{majrtt}_i - 2\varepsilon$, thus proving the result. □

[1]The results can be adapted for clocks with skew.

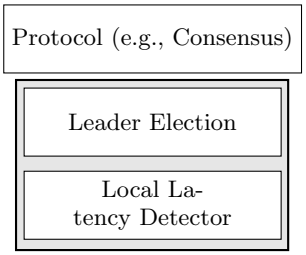

Figure 1: The three layers of the system

2.2 Optimal Leader Election

The classical definition of leader election in the literature is the following:

DEFINITION 1 (LEADER ELECTION PROBLEM). *There exists a correct process ℓ and a time t after which, for every alive process p, $leader_p = \ell$.*

We need to extend this definition with the notion of optimality. Since in our model, the variations even after GST allow the optimal leader to change, the notion of an *optimal* leader is somehow contradictory to the definition of a leader. However, as we show later, we can guarantee that the elected leader is within a small interval from optimality:

DEFINITION 2 (σ-DELAY-OPTIMAL PROCESS AT TIME t). *A process p is* σ-delay-optimal *at time t if there is no other process q that has a majority latency $majrtt_q(t)$ at time t that is smaller than $majrtt_p(t) - \sigma$.*

Note that there might be several σ-delay-optimal processes. The goal is to elect a leader among these.

DEFINITION 3 (σ-OPTIMAL LEADER). *There exists a correct process ℓ and a time t after which, for every alive process p, $leader_p = \ell$ and ℓ is σ-delay-optimal for all $t' \geq t$.*

3. σ-OPTIMAL LEADER ELECTION

We consider a two-layer implementation of our oracle as depicted in Figure 1. The local latency detector layer provides to the leader election protocol a vector with the estimation of the RTT times from the local process to all other processes in the system. The leader election algorithm uses this information when electing a new leader.

It is important to notice that the leader election module uses the information of the latency detector only to optimize the choice of leader but not to detect faulty leaders. The latter is done by the leader election algorithm, where the current leader sends heartbeat messages to every other process. The reason for this approach is that measuring the latencies of the system is quite costly: it requires $O(n^2)$ messages in general, while a communication efficient implementation of the leader election algorithm requires only $O(n)$ links to carry the periodical heartbeat messages. Separating these two issues, allows implementations to control the tradeoff between the number of messages sent and the reaction time to latency changes: faster reaction times can be obtained at a cost of increased network load by measuring the latencies more frequently. Implementations can therefore be tuned for each specific case, taking in consideration the underlying network and the QoS requirements.

3.1 The Local Latency Detector

The lowest layer of our system is the *local latency detector*. A local latency detector module periodically outputs an n-dimensional vector L_i at each process i, where $L_i[j]$ represents the estimate of the round-trip time between process i and process j. In contrast to previous approaches [7], there is no global oracle that provides global latency information to every process, but every process gets only an estimate of the round-trip delays to its peers. However, since even after GST round-trip delays may change, the output of the latency detector has only limited accuracy.

The implementation of such an oracle is done by sending every η time ping-pong messages between all processes to measure round-trip times. The sampling period η is a configuration parameter that controls the tradeoff between network load (*i.e.*, the number of messages per time period) and speed of adaptation of the algorithm to changes in the latency of the links. Then, from the system model we get the following property:

LEMMA 2 (LOCAL LATENCY DETECTOR). *Consider a latency detector that is implemented by every process sending pings to all every η time. Then we have*

$$\forall t > GST + \eta + 2\Delta, \forall i, j : |L_i[j] - \overline{rtt}_{ij}(t)| \leq 2\varepsilon.$$

PROOF. After GST, the duration rtt of a round-trip which started at some time t is bounded by $|rtt - \overline{rtt}_{ij}(t)| \leq 2\varepsilon$. Since each round-trip takes at most 2Δ and measurements are done every time η, then the latest by $GST + \eta + 2\Delta$ the output of the detector is bounded. □

Note that GST and $\overline{rtt}_{ij}(t)$ are unknown system model parameters, ε is a known system model parameter, and η is an implementation-dependent parameter of the latency detector.

3.2 Electing a leader

The leader election algorithm is given as Algorithm 1 and Algorithm 2. In the following subsections, we explain the three different ideas of the algorithm separately, although in the algorithm they work closely together. The proof of correctness is given in Section 4.

3.2.1 Leader Election

The leader election part of the algorithm is based on an algorithm from [1]. The algorithm is organized as a sequence of rounds, with the leader of round r being process r mod n. Processes remain in round r as long as they believe process r mod n should remain the leader, that is, it is correct and optimal. The leader p sends ALIVE messages to all processes every 2Δ time. If a process does not receive any message from the leader for more than 3Δ time, it suspects the leader to have failed and advances to the next round $r+1$ (lines 38–40). It then sends a (START, r_p) message with $r_p = r + 1$ to the leader of round $r + 1$ (line 9). Upon receiving this message, the new leader advances to the new round (lines 17–18) and sends ALIVE messages to all. This will force all other processes to advance to round $r + 1$.

The selection of an optimal leader is done in lines 32, 36–37, where the current leader calls *SelectLeader*. This function encapsulates the election of an optimal leader, based on *rttMatrix*. If it returns a process different than the current leader, then the current leader advances to a higher round, forcing the election of the new process. *SelectLeader* is independent from the rest of the algorithm, and is explained in Section 3.2.3.

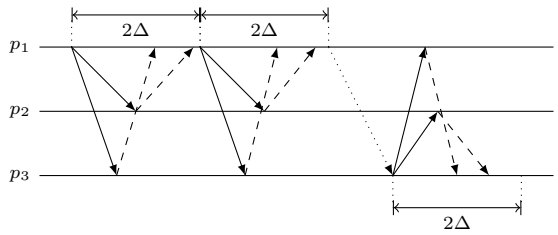

Figure 2: Messages exchanged by the algorithm, where solid arrows represent ALIVE messages, dashed arrows represent REPORT messages, and dotted arrows represent START messages.

3.2.2 Aggregating RTT information

In order to elect a leader, local knowledge about latencies, *i.e.*, the output of the latency detector is not sufficient. In contrast to the approach in [7], we do not implement a global oracle but the latency information is aggregated only at the current leader.

Every 2Δ time, the leader sends ALIVE messages (lines 15, 32–34) and resets the latency matrix *rttMatrix* (line 13). The other processes reply with a REPORT message (line 25). The leader uses these messages to rebuild *rttMatrix* (line 19). This way, 2Δ after sending ALIVE, the *rttMatrix* matrix of the leader contains estimates that are no older than 2Δ. If the ALIVE messages are sent after $GST+\eta+2\Delta$, by Lemma 2 the *rttMatrix* is such that $\left|rttMatrix[i][j] - \overline{rtt}_{ij}\right| \leq 2\varepsilon$.

The REPORT message is sent after every ALIVE received, even if the local rtt vector did not change. Rebuilding the *rttMatrix* from scratch every 2Δ ensures that if some process i crashes, after the next ALIVE cycle, for all j, we have $rttMatrix[i][j] = \infty$. Note that if old values were kept until updated by a REPORT message, $rttMatrix[i]$ would never be updated.

Figure 2 illustrates the algorithm. Initially p_1 is the leader. After the first cycle, the *rttMatrix* built with the ALIVE messages from the other processes still indicates that p_1 is the most suitable process to be the leader, so p_1 sends a new cycle of ALIVE messages. In the meanwhile, new measurements were taken by some process and the *rttMatrix* built by p_1 at the end of the second cycle shows that p_3 is now better suited to be the leader. So p_1 advances to the next round where p_3 is the leader and sends a START message to p_3. Upon receiving it, p_3 advances rounds, assumes leadership and starts sending ALIVE messages.

3.2.3 Selecting a σ-optimal leader

The protocol explained so far ensures that in a good period (i) no faulty process is elected as leader and (ii) the current leader periodically calls *SelectLeader* to verify whether it is still the σ optimal one.

We describe now *SelectLeader*, see Algorithm 2. Note that from Section 3.2.2, the leader gets a matrix L for which Lemma 2 holds. *SelectLeader* starts by ordering each line of the matrix by increasing latencies. As a result, for all i, element $(i, \lfloor n/2 \rfloor + 1)$ of the matrix gives an estimate of the RTT of a majority of processes to process i.

Algorithm 1 Delay-aware leader election.

```
1:  Initialization:
2:    ∀p : localrtt_p[p] ← ∞
3:    r_p ← 1
4:    StartRound(r_p)

5:  procedure StartRound(r)
6:    r_p = r
7:    leader = r_p mod n
8:    reset timer to Δ
9:    send (START, r_p) to q
10:   if p = leader then
11:     SendAlives()

12: procedure SendAlives()
13:   ∀p, q : rttMatrix_p[p][q] ← ∞
14:   reset timer
15:   send (ALIVE, r_p) to all processes except p

16: upon receive (REPORT, r, rtt) from q with r ≥ r_p
17:   if r > r_p then
18:     StartRound(r)
19:   rttMatrix_p[q] = rtt

20: upon receive (ALIVE, r) from q
21:   if r < r_p then
22:     send (START, r_p) to q
23:   if r = r_p then
24:     reset timer
25:     send (REPORT, r_p, localrtt_p) to leader
26:   if r > r_p then
27:     StartRound(r)

28: upon receive (START, r) with r > r_p
29:   StartRound(r)

30: upon timer = 2Δ
31:   if p = leader then
32:     newLeader ← SelectLeader(rttMatrix_p, leader)
33:     if newLeader = leader then
34:       SendAlives()
35:     else
36:       r ← smallest k > r_p s.t. k mod n = p
37:       StartRound(r)

38: upon timer = 3Δ
39:   if p ≠ leader then
40:     StartRound(r_p + 1)

41: upon new rttVector from latency detector
42:   localrtt_p ← rttVector
```

Algorithm 2 Choosing a leader.

```
1: procedure SelectLeader(L, leader)
2:   for all lines i ∈ {1, ..., n} do
3:     sort L[i] by increasing latencies
4:   curRtt ← L[leader][⌊n/2⌋ + 1]
5:   minRtt ← min_{i∈{1,...,n}}{L[i][⌊n/2⌋ + 1]}
6:   if minRtt < curRtt − 4ε then
7:     return min{i ∈ {1, ..., n} | L[i][⌊n/2⌋ + 1] = minRtt}
8:   else
9:     return leader
```

The key point here is that a new leader is elected only if its majority-RTT is 4ε better than the current one (lines 6–7). Since by Lemma 1 the majority-RTTs of processes vary at most by $\pm 2\varepsilon$ after GST, this prevents that the role of the leader oscillates between two processes. Moreover, as shown in the proofs, the selection rule ensures that if the leader changes, the new leader is a process with a lower $\overline{majrtt}$. Thus, even if the leader changes after GST, this happens only a finite number of times (the $\overline{majrtt}$ are fixed), and thus eventually a single correct process is leader forever. This process is an 12ε-optimal leader:

PROPOSITION 1. *Algorithms 1 and 2 solve the σ-optimal leader election problem with $\sigma = 12\varepsilon$.*

4. PROOF OF CORRECTNESS

We first prove that after $GST + \eta + 2\Delta$ the function *SelectLeader* returns only processes whose $\overline{majrtt}$ is within 8ϵ of the optimal. Then, we prove that after GST once a leader is elected the leadership will change only to another process that has a lower $\overline{majrtt}$. Therefore, the number of possible changes is limited and the system eventually elects a leader forever that is within 12ϵ of the optimal.

For the proofs below, let p_m denote the process with minimal $\overline{majrtt}_{p_m}$; if there are several such processes, the one with the lowest id.

DEFINITION 4. $\mathcal{O} = \{p \in \Pi \mid \overline{majrtt}_p \leq \overline{majrtt}_{p_m} + 8\varepsilon\}$

Since for $t > GST$, $|majrtt_{p_m}(t) - \overline{majrtt}_{p_m}| \leq 2\varepsilon$, we have:

COROLLARY 1. *Every process from $\mathcal{O}$ is 12ε-optimal at any time after GST.*

LEMMA 3. *The set $\mathcal{O}$ is non-empty and contains only correct processes.*

PROOF. Trivially, the set is non-empty because $p_m \in \mathcal{O}$. By definition, if p has crashed, then for all process j, $\overline{\delta}_{pj} = \infty$ and thus $\overline{majrtt}_p = \infty$. Therefore, $p \notin \mathcal{O}$. □

LEMMA 4. *Let M a matrix with $\left|M[i][j] - \overline{rtt}_{ij}\right| \leq 2\varepsilon$ for all $i, j \in \Pi$. Then SelectLeader(M, ℓ) returns a process p such that: (i) $p \in \mathcal{O}$ and (ii) if $p \neq \ell$ then $\overline{majrtt}_p < \overline{majrtt}_\ell$.*

PROOF. For (i), lines 2 and 3 of Algorithm 2 order each line of M in ascending order. Line 5 computes *minRtt*, which is the minimal majority value among the estimations contained in M. Let q be the process with the lowest id such that $M[q][\lfloor n/2 \rfloor + 1] = minRtt$.

From the assumption on M, we have $M[p_m][\lfloor n/2 \rfloor + 1] \leq \overline{majrtt}_{p_m} + 2\varepsilon$. And since $M[q][\lfloor n/2 \rfloor + 1] \leq M[p_m][\lfloor n/2 \rfloor + 1]$ (because of the definition of q), we get

$$M[q][\lfloor n/2 \rfloor + 1] \leq \overline{majrtt}_{p_m} + 2\varepsilon \tag{1}$$

If the condition at line 6 is true, then *SelectLeader* returns q and by (1) we have that $q \in \mathcal{O}$. Else, if the condition is false, *SelectLeader* returns ℓ, and we have $curRtt \leq minRtt + 4\varepsilon$, which is equivalent to:

$$M[\ell][\lfloor n/2 \rfloor + 1] \leq M[q][\lfloor n/2 \rfloor + 1] + 4\varepsilon. \tag{2}$$

By assumption, we have $\overline{majrtt}_\ell - 2\varepsilon \leq M[\ell][\lfloor n/2 \rfloor + 1]$. Using this inequality on the left side of (2) and (1) on the right side, we obtain $\overline{majrtt}_\ell - 2\varepsilon \leq \overline{majrtt}_{p_m} + 6\varepsilon \Leftrightarrow \overline{majrtt}_\ell \leq \overline{majrtt}_{p_m} + 8\varepsilon$. Therefore, $\ell \in \mathcal{O}$.

To prove (ii), since $p \neq \ell$, the condition on line 6 is true, that is, $minRtt < curRtt - 4\varepsilon$, which is equivalent to:

$$M[p][\lfloor n/2 \rfloor + 1] < M[\ell][\lfloor n/2 \rfloor + 1] - 4\varepsilon \tag{3}$$

By assumption, we have $\overline{majrtt}_p - 2\varepsilon \leq M[p][\lfloor n/2 \rfloor + 1]$ and $M[\ell][\lfloor n/2 \rfloor + 1] \leq \overline{majrtt}_\ell + 2\varepsilon$. Using these two inequalities in (3), we obtain $\overline{majrtt}_p - 2\varepsilon < \overline{majrtt}_\ell + 2\varepsilon - 4\varepsilon$ and thus $\overline{majrtt}_p < \overline{majrtt}_\ell$. □

LEMMA 5. *After time GST+η+4Δ, if SelectLeader*(M, ℓ) *is called,* $\left|M[i][j] - \overline{rtt}_{ij}\right| \leq 2\varepsilon$ *for all* $i, j \in \Pi$.

PROOF. Let ℓ be the process that calls *SelectLeader*(M, ℓ) at time $t \geq GST+\eta+4\Delta$ while in round r_ℓ. By Algorithm 1, ℓ is the leader for round r_ℓ and the last time it sent ALIVE messages was at time $t-2\Delta$. At this time it reset all entries of the matrix M to ∞. The ALIVE message is received by all processes during the interval $[t-2\Delta, t-\Delta]$. All alive processes answer with a REPORT message containing their local vector. These messages are received by ℓ before time t and are used to update the matrix M. If a process q is crashed when it received the ALIVE message, then line $M[q]$ is not updated and remains equal to a vector of ∞.

Thus, the matrix M contains values that are no older than 2Δ. Since $t - 2\Delta = GST + \eta + 2\Delta$, by Lemma 2 we have $\left|M[i][j] - \overline{rtt}_{ij}\right| \leq 2\varepsilon$ for all $i, j \in \Pi$. □

From (ii) of Lemma 4 and Lemma 5, and the fact that we have a finite number of processes, we get:

COROLLARY 2. *After* $GST + \eta + 4\Delta$, *the leader changes only a finite number of times.*

Next we show that if no new round is started after GST, then the optimal leader is elected. The proof of Proposition 1 discusses the other case, when new rounds are started after GST.

LEMMA 6. *If there is a time t after which the maximum round reached by any process does not increase, then eventually a process from* $\mathcal{O}$ *is elected as leader.*

PROOF. This proof is in two parts: (i) show by contradiction that a leader is elected, (ii) show that this leader must be in $\mathcal{O}$.

Let us assume that no leader is ever elected. Let r_h be the highest round of any process after t. Other processes might advance to higher rounds if they are in lower rounds but, by assumption, will never exceed r_h. Since by assumption processes cannot increase rounds forever, there is a time t' after which no process advances to a new round anymore.

Let $\mathcal{H}$ be the set of processes in round r_h after time t'. Let p be the candidate leader for round r_h (*i.e.*, $p \bmod n = r_h$).

If $p \in \mathcal{H}$, then p sends ALIVE messages to all processes every 2Δ. Therefore, at most at $t' + 3\Delta$, all processes will have received an ALIVE message for round r_h. Since by assumption no process is in a round higher than r_h, all alive processes will accept this message and advance to round r_h if not already there. Therefore, p becomes the leader for all alive processes, contradicting the assumption that no leader is elected.

If $p \notin \mathcal{H}$, then processes in $\mathcal{H}$ will timeout waiting for the ALIVE messages from the leader and will advance to a higher round, contradicting the assumption that no process advances to a new round after time t'.

To show (ii), let p be the process elected as leader. By assumption, p remains leader forever and keeps sending ALIVE messages every 2Δ time. Let t^* be the first time that p sends ALIVE messages after $GST + \eta + 2\Delta$. We can apply Lemmas 4 and 5 to show that by time $t^* + 4\Delta$ a process $\ell \in \mathcal{O}$ is elected leader. If $\ell \neq p$, then p advances to a higher round which contradicts the fact that p remains leader forever. Therefore, $\ell \in \mathcal{O}$. □

The following lemma shows that after GST, once there is a leader ℓ, the leader will only change as a result of ℓ calling *SelectLeader*. Together with Lemmas 4 and 5 this shows that the leader only changes to processes with better performance.

LEMMA 7. *Let* ℓ *be a process that at time* $t > GST$, *is in round* r_ℓ *for which* ℓ *is the leader. Let* t' *be the time when* ℓ *sends the next* ALIVE *message. If all alive processes are in a round not higher than* r_ℓ *when they receive the* ALIVE *message from* ℓ, *then (i) no process* $q \neq \ell$ *advances to a round* $r' > r_\ell$ *while* ℓ *remains in round* r_ℓ, *and (ii)* ℓ *only advances to a new round if* *SelectLeader*$_\ell(M, \ell)$ *returns a process different than* ℓ.

PROOF. To prove (i), note that since we are after GST the ALIVE message sent by ℓ at time t' is ready for reception at q the latest by time $t' + \Delta$. By assumption, at this time q is in a round not higher than r_ℓ. If q is in a lower round, it advances to r_ℓ. We now show that if ℓ remains in round r_ℓ, then q does not advance to a round higher than r_ℓ.

We proceed by contradiction. Let q be the first process to advance to a round higher than r_ℓ. Then either (a) q received a higher round message, (b) the timer of q expired, or (c) q called the procedure *SelectLeader*. (a) cannot happen because q is the first process to advance to a round higher than r_ℓ. (b) is not possible either, because the ALIVE messages from ℓ are received always with less than 3Δ of interval. Finally, q cannot call *SelectLeader* because it is not the leader for round r_ℓ.

To prove (ii), note that since ℓ is the leader, it will never timeout on itself. Additionally, by (i) we know that no other process will advance to a new round before ℓ, so ℓ will never receive a higher round message. Therefore, the only other way in which ℓ can advance to new rounds is if *SelectLeader*$_\ell(M, \ell)$ returns a process different than ℓ. □

LEMMA 8. *Assume there is a process p and a round* r_p *such that p is the first process starting round* r_p *at time* t_0, $t_0 > GST + 4\Delta$. *Then before time* $t_0 + 2\Delta$ *no process will advance to a round* $r' > r_p$.

PROOF. We proceed by contradiction. Let us assume there's a process $q \neq p$ that is the first to start a round $r' > r_p$ at time t', with $t_0 < t' < t_0 + 2\Delta$. This could have happened in one of the following ways: (i) receiving a message from a higher round, (ii) timeout on ALIVE messages from the leader, and (iii) calling *SelectLeader*.

Case (i) is impossible, because q is the first process starting round r'. In case (ii), q advances from round $r' - 1$ to round r' when this timeout occurs. By Algorithm 1, q must have been in round $r' - 1$ for at least 2Δ, which means that q started round $r' - 1$ before time t_0. Since $r' - 1 \geq r_p$, we get a contradiction with the assumption that p is the first process starting round r_p.

Finally, for (iii), let's assume that q advanced at time t' because of calling *SelectLeader*. This means that q was in a round $r_q < r'$ for which q is the leader, and after calling *SelectLeader* it advanced to round r'. We'll show that this contradicts the assumption that p was the first process to advance to round r_p at time t_0.

Process q was in round r_q for at least 2Δ, which is the time a process waits after entering a round until calling *SelectLeader* for the first time. Therefore, round r_q was started no later than $t' - 2\Delta$. And since $t_0 < t' < t_0 + 2\Delta$, we have $t' - 2\Delta < t_0 < t'$. From the above, it comes immediately that $r_q < r_p$, because by assumption no process started a round equal or higher than r_p before t_0.

Since q is the leader for round r_q, it sends ALIVE messages every 2Δ while in round r_q. Since $t' - 2\Delta$ is the latest that q entered the round, q sent an ALIVE message between $t' - 4\Delta$ and $t' - 2\Delta$. Since $t' - 4\Delta > GST$, this message is received by all alive processes between $t' - 4\Delta$ and $t' - \Delta$. If there is any process in a round higher than r_q when it receives the ALIVE from q, it answers with a $(START, r)$ message, which is received before t', forcing q to advance to a new round, which contradicts the assumption that q calls *SelectLeader* at time t'. Therefore, all processes are in a round $r \leq r_q$ when they receive the ALIVE message from q. Applying Lemma 7, we conclude that no process will advance to a new round until q calls *SelectLeader*. But by assumption the next time this happens is at time t', contradicting the assumption that p advances to round r_p at time $t_0 < t'$. □

PROPOSITION 1. *Algorithms 1 and 2 solve the σ-optimal leader election problem with $\sigma = 12\varepsilon$.*

PROOF. We prove that eventually a permanent leader $\ell \in \mathcal{O}$ is elected. Then, by Corollary 1 it comes that ℓ is 12ε-optimal.

Let $t_s = GST + \eta + 4\Delta$. If after t_s no new round is started, then by Lemma 6, a process $\ell \in \mathcal{O}$ is eventually elected. We now prove the other case, where some new round is started after t_s. Assume there is a process p and a round r_p such that p is the first process starting round r_p after t_s at time t_0. We consider two cases: (i) p is the leader for round r_p and (ii) p is not the leader for round r_p.

Starting by (i), p sends ALIVE messages to all processes at time t_0. These messages are received the latest by time $t_0 + \Delta$. By Lemma 8 no process is in a higher round at this time. We can now use Lemma 7 to conclude that no process advances to a new round unless a call to *SelectLeader* made by p returns a process different than p. Since $t_0 \geq GST + \eta + 4\Delta$, by Lemmas 4 and 5 we know that all future invocations of *SelectLeader* will return a process $\ell \in \mathcal{O}$. If $p \notin \mathcal{O}$ then the first invocation of *SelectLeader* by p will return a process $\ell \in \mathcal{O}$, making p pass leadership to ℓ. Otherwise, p may or may not remain leader forever, but if the leader changes, it will always be to a process in $\mathcal{O}$. Either way, by Corollary 2, the leader can only change a finite number of times after t_0 and eventually a σ-optimal process is elected permanently.

In case (ii), p sends a START message to the leader q of round r_p and waits for 2Δ (Line 8 sets the timer to Δ and the timer expires at 3Δ for non-leader processes). Process q may either be alive or crashed. If q is alive, then by time $t_0 + \Delta$, q receives the message and advances to round r_p. At the same time it sends ALIVE messages to all, which arrive by time $t_0 + 2\Delta$. By Lemma 8, no process is in a higher round at this time. We are now in the same conditions as in case (i), so the same reasoning applies. If q is crashed, p will timeout at time $t_0 + 2\Delta$ and advance to round $r_p + 1$. Since by Lemma 8, no process other than p advances to a round higher than r_p before $t_0 + 2\Delta$, p is the first in round $r_p + 1$. We are again on the conditions of this Lemma, so we can apply the same reasoning to conclude that p will try to contact the leader for round $r_p + 1$. Since there are a maximum of f crashed processes, eventually p will reach a round where the leader q is alive and we can apply the same reasoning as above to conclude that a σ-delay-optimal leader is elected. □

5. CONCLUSION

In this paper, we have given a model where the latencies of all links between correct processes eventually stabilize, so that they are within an interval of $\pm\varepsilon$ around some fixed value. We have given and proved correct an algorithm that elects as leader a process that is 12ε-optimal after the global stabilization time.

From a practical point of view, for algorithms like Paxos [5], a leader needs to be the same only for a bounded time interval to solve consensus. In such a system, the parameter ε trades stability against optimality: with a small value of ε, the algorithm adapts to smaller changes in the network and thus selects processes that are closer to the optimal, risking, however, frequent leader changes. A large value of ε on the other hand favors long-lived leaders over optimality.

6. REFERENCES

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Proceedings of the 15th International Conference on Distributed Computing (DISC'01)*, pages 108–122. Springer-Verlag, 2001.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega with weak reliability and synchrony assumptions. In *Proceeding of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'03)*. ACM Press, 2003.

[3] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proceeding of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC'04)*, pages 328–337, St. John's, Newfoundland, Canada, 2004. ACM Press.

[4] M. Hutle, D. Malkhi, U. Schmid, and L. Zhou. Brief announcement: Chasing the weakest system model for implementing Ω and consensus. In *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS '06)*, 2006.

[5] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[6] D. Malkhi, F. Oprea, and L. Zhou. Ω meets paxos: Leader election and stability without eventual timely links. In *Proceedings of the 19th International Conference on Distributed Computing (DISC'05)*, pages 199–213, 2005.

[7] L. Sampaio and F. Brasileiro. Adaptive indulgent consensus. In *Dependable Systems and Networks (DSN 2005)*, pages 422–431, Yokohama, Japan, 2005. IEEE Computer Society.

[8] L. Sampaio, R. C. Nunes, F. Brasileiro, and I. Jansch-Pôrto. Efficient and robust adaptive consensus services based on oracles. *Journal of the Brazilian Computer Society (JBCS), Special Issue on Dependable Computing (2005)*, 2005.

[9] L. M. R. Sampaio, F. V. Brasileiro, W. Cirne, and J. C. A. Figueiredo. How bad are wrong suspicions? towards adaptive distributed protocols. In *Dependable Systems and Networks (DSN 2003)*, pages 551–560, Los Alamitos, CA, USA, 2003. IEEE Computer Society.